Assignment: Weighted Interval Scheduling

Weighted Interval Scheduling is a generalization of interval scheduling where a value is assigned to each task and the goal is to maximize the total value of the set of compatible tasks. The greedy algorithm analyzed in class no longer guarantees an optimal solution. (You should find a counter example to convince yourself this is true.) You assignment is to develop a Dynamic Programming algorithm to solve the Weighted Interval Scheduling problem. As usual the main steps in developing a solution to the problem are to:

- **A.** **Specify the function that represents the quantity to be optimized**. In this case let ValWIS (??) represent the maximum total value of a Weighted Interval Scheduling problem of a given size specified by the parameter(s).
- **B.** Derive the **recurrence relation** that describes ValWIS (??) in terms of smaller problem instances. Don't forget to specify the base case(s).
- **C.** The **specification of the table** that you would use in a bottom up programmatic solution. What are the dimensions of the table and what does each represent.
- **D.** The **specification of the algorithm** for filling in the table that you would use in a bottom up programmatic solution. That is convert the recurrence relation (**B.**) to an iterative algorithm.
- **E.** The **specification of the algorithm** for tracing back through the table to find the set of compatible tasks that gives the maximum total value.
- **F.** The derivation of the closed form solution of the complexity of filling in the table.

Submit a file to PolyLearn: A Java class *WgtIntScheduler.java*. The class WgtIntScheduler.java should contain a public method **public static int[] getOptSet (int[] stime, int[] ftime, int[] weight)**

- The input represents the start times, finish times, and weights of jobs 1 .. n – indexed from 0 to n-1 in the arrays. The return array contains only **idnumbers** of the intervals for the optimal solution. Note the idnumbers are just (1+ index of the job in the input array). In test case 1 below **job 3** starts at time 2, ends at time 6 and has weight 5. Note that the jobs input may not be in any particular order. Thus if your algorithm requires the jobs to be ordered you must sort the jobs appropriately keeping track of their idnumbers.

getOptSet returns the jobs that make up the optimal set **in increasing order of their idnumbers**.

Input arrays – test case 1

```
{4,  3,  2, 10, 7}          // start times for jobs 1, 2, 3, 4, 5
{7, 10, 6, 13, 9}           // finish times for jobs 1, 2, 3, 4, 5
{6,  6,  5,  2,  8}         // weights for jobs 1, 2, 3, 4, 5
```

Returns:

- array of size three, containing 1, 4, 5

Input arrays – test case 2

```
{3,  3,  1, 10, 8}          // start times for jobs 1, 2, 3, 4, 5
{7, 10, 4, 13, 11}          // finish times for jobs 1, 2, 3, 4, 5
{6,  9,  5,  8, 10}         // weights for jobs 1, 2, 3, 4, 5
```

Returns:

- array of size two, containing 2, 4